Practical Performance Model for Optimizing Dynamic Load Balancing of Adaptive Applications[*]

Kevin Barker
CCS-3 Modeling, Algorithms, and Informatics
Los Alamos National Laboratory
Los Alamos, NM 87545
kjbarker@lanl.gov

Nikos Chrisochoides
Computer Science Department
College of William and Mary
Williamsburg, VA 23187
nikos@cs.wm.edu

Abstract

Optimizing the performance of dynamic load balancing toolkits and applications requires the adjustment of several runtime parameters; however, determining sufficiently good values for these parameters through repeated experimentation can be an expensive and prohibitive process. We describe an analytic modeling method which allows developers to study and optimize adaptive application performance in the presence of dynamic load balancing. To aid tractibility, we first derive a "bi-modal" step function which simplifies and approximates task execution behavior. This allows for the creation of an analytic modeling function which captures the dynamic behavior of adaptive and asynchronous applications, enabling accurate predictions of runtime performance. We validate our technique using synthetic microbenchmarks and a parallel mesh generation application and demonstrate that this technique, when used in conjunction with the PREMA runtime toolkit, can offer users significant performance improvements over several well-known load balancing tools used in practice today.

1. Introduction

Although many techniques have been developed for the load balancing of loosely synchronous parallel applications, providing such support for *asynchronous* and *adaptive* codes is still an open problem. In response, we have developed the Parallel Runtime Environment for Multicomputer Applications (PREMA) [4, 3], a mid-level software toolkit for the dynamic load balancing of this challenging class of applications. However, in order to make the most effective use of the runtime software, and subsequently maximize the utilization of computational resources, certain parameters governing PREMA's execution must be set off-line. While optimal values for these variables can be determined through repeated executions of the target application, such a procedure is time-consuming, potentially expensive, and often prohibitive for large systems and applications. We present an analytic model that allows the user, given a set of assumptions such as communication latency and coarse bounds to task execution times, to quickly predict application performance for codes built using the PREMA library. With this tool, developers can fine-tune the performance of applications built using PREMA in an inexpensive, off-line manner.

However, accurately modeling the dynamic load balancing behavior of a given set of tasks is a difficult problem. Therefore, we define an approximate problem, using a "bi-modal" task distribution, which serves as an accurate abstraction of the original task set. This approximation consists of only two task types and can therefore be tackled analytically.

Once an appropriate bi-modal approximation is derived, the modeling technique described can be used to estimate upper and lower bounds on an application's runtime in the presence of dynamic load balancing. After validating the model's accuracy, we conduct a parametric study to quantitatively analyze the impact of two important runtime parameters: task granularity (or level of *over-decomposition*) and the preemption *quantum*, or interval after which the runtime system will preempt ongoing computation to receive and process load balancing messages. Both parameters represent a tradeoff between system and communication induced overhead and load balancing flexibility. We demonstrate that the PREMA software, in conjunction with the analytic modeling capability provided by this technique, will allow users to easily configure parallel applications that are capable of out-performing codes using other load balanc-

[*] This work was supported by NSF Career Award #CCR-0049086, ITR #ACI-0085969, RI #EIA-9972853, NGS #ANI-0203974, and ITR #CNS-0312980

ing tools available to the community.

2. PREMA Runtime Environment

The modeling technique we present is designed to predict the performance of applications utilizing the dynamic load balancing capability of the PREMA runtime environment; therefore, a brief description of PREMA is warranted. Applications begin by decomposing the data domain into *mobile objects*, which are registered with the runtime system. Choosing a greater number of mobile objects than available processors is referred to as *over-decomposition*; a greater degree of over-decomposition will allow for more load balancing flexibility at the cost of some overhead. Computation is invoked via *mobile messages*, which are addressed to mobile objects themselves, not to the processors on which the objects reside. This allows mobile objects to migrate from processor to processor without needing to explicitly notify the application; the PREMA runtime system is responsible for efficiently routing messages to their correct destinations. As mobile objects migrate between processors, any pending computation moves as well; mobile objects are therefore the units of load balancing granularity. Migrating data thereby implicitly migrates computation.

PREMA provides a load balancing framework through which a wide variety of load balancing algorithms may be implemented. In a typical policy, such as the Diffusion [11, 2] method, load balancing begins when a processor's local work load falls below a pre-defined threshold. Requests for tasks are sent to neighboring processors; if a neighbor has a sufficient number of tasks (mobile objects with pending computation) available, one will be uninstalled and migrated to the requesting processor.

Each processor executes both the application thread and a separate *polling thread*, whose job is to periodically awaken and probe the network for load balancing requests. The *quantum*, or period of time between thread awakenings, can be set by the user but remains static throughout the application's execution. By handling load balancing requests within a seperate thread, PREMA is able to dramatically shorten the delay between migration request and response, relative to single-threaded load balancing libraries, and thereby decrease the number of idle cycles and application execution time.

3. Approximation of Task Execution Times

Accurately modeling a general distribution of task weights is a challenging problem. To simplify it and make the problem tractible, we model the estimated task cost function ($task\_weight = f(task\_id)$) using a bi-modal (or step) function that defines two classes of equally weighted tasks. Sorting tasks in terms of their weights into a monotonically increasing order allows us to define a parameter *Gamma* (Γ) which divides the task pool into heavier tasks, which are termed *Alpha* (α), and lighter tasks, which are termed *Beta* (β). Because the applications we target are adaptive, precise task weights are not always known in advance. Therefore, approximate weights can be used as inputs to the model; however, the more accurately task weights are known, the more accurate the model's predictions will be.

We are able to define a unique approximation function using the following two criteria:

1. The area under the step function must be equal to the area under the curve defined by the original cost function. This is equivalent to stating that the amount of computation invoked by the tasks in the original cost function must equal the computation invoked by the approximation model.
2. The computational complexity of the tasks contained within the α and β approximation classes must "accurately" reflect the weights of the tasks in the original cost function (we define this precisely below). This ensures that the model accurately predicts the time at which load balancing begins and the amount of computation migrated during each load balancing operation.

$$Work_{\alpha} = \sum_{i=\Gamma+1}^{N} T_i = (N-\Gamma) \times T_{\alpha\_task} \tag{1}$$

$$Work_{\beta} = \sum_{i=1}^{\Gamma} T_i = \Gamma \times T_{\beta\_task} \tag{2}$$

$$Work_{Total} = \sum_{i=1}^{N} T_i = Work_{\alpha} + Work_{\beta} \tag{3}$$

For a given selection of Γ, there are unique values of α and β task execution times (T_{α_task} and T_{β_task}) that satisfy Equations 1, 2, and 3 (in which T_i is the computational weight, or time required by task i, and N is the number of tasks). The selection of Γ, from the N possible choices, is determined from the second criterion. A unique Γ minimizes the sum of $Error_{\alpha}$ and $Error_{\beta}$, defined by Equations 4 and 5[1]. Each error term is a measure of the accuracy (as in least-square approximation [14]) in which the selection of the approximation task weights represents the original cost function.

$$Error_{\alpha} = \sum_{i=\Gamma+1}^{N} (T_{\alpha\_task} - T_i)^2 \tag{4}$$

1 In the case in which all tasks are of equal weight, Γ is not unique; however this case requires no load balancing and so is not considered further.

$$Error_{\beta} = \sum_{i=1}^{\Gamma} (T_{\beta\_task} - T_i)^2 \qquad (5)$$

4. Analytic Modeling for Diffusion Load Balancing

We next apply our modeling technique to the Diffusion [11, 2] load balancing method, which can be trivially extended to include the Work-stealing [2] method. In our research, we have found these two methods to be the most generally applicable to a wide variety of problem types.

Our approach is to model the runtime of the slowest processor (which we term the *dominating* processor), as this will determine the overall runtime of the parallel application. Computation will progress with each processor consuming its initially allocated tasks. The processors initially assigned β tasks will finish their computation first, at which point they begin requesting tasks from other processors. The *polling thread*, incorporated into the architecture of the PREMA runtime system, will awaken on each processor after a specified *quantum* of time in order to process any pending load balancing requests. If a request is received and sufficient tasks are currently in the local work pool, an α task which has not yet begun execution will be migrated to the requesting processor.

$$T_{total} = T_{work} + T_{thread} + T_{comm}^{app} + T_{comm}^{lb} + T_{migr}^{lb} + T_{decision}^{lb} - T_{overlap} \qquad (6)$$

Once the requisite task partitioning information is defined, Equation 6 is used to predict application runtime. It provides a model of runtime on a single processor composed of computation invoked by application tasks, runtime system overhead, communication, and task migration. Evaluating Equation 6 from the point of view of initially overloaded (α) and initially underloaded (β) processors allows us to determine the dominating processor type, and hence overall run time.

4.1. Computation Component

The T_{work} term of Equation 6 encompasses the amount of time attributable to task execution on a single processor, taking into consideration task migration due to dynamic load balancing. Load balancing will begin once all β tasks have completed, at time $T_{\beta} = \frac{N}{P} \times T_{\beta\_task}$ (our model assumes that each of P processors is initially assigned an equal fraction of the N tasks). At this point, a suitable task for migration must be located; inquiries are sent to an evolving set of neighboring processors until an α task is found. In the best case, this will require a single request. In the worst case, all comparably underloaded nodes will be probed before a suitable task is located[2]. For simplicity, we use the term T_{locate} to describe the time required for task location; the time required for each migration request is defined in Section 4.4.

Once T_{β} is known, the next step is to calculate the number of tasks potentially available for migration. We define $T_{\alpha} = \frac{N}{P} \times T_{\alpha\_task}$ to be the time required to complete all α tasks, barring migration. This enables the specification of $T_{\triangle} = T_{\alpha} - T_{\beta} - T_{locate}$, which defines the amount of work available for migration. The number of tasks available for load balancing can be calculated from $T_{\triangle}$ and T_{α_task}. Because of the upper and lower bounds on T_{locate}, there will be upper and lower bounds on the number of migratable tasks, and therefore bounds on the application's execution time.

To calculate the number of task migrations, we first determine the number of load balancing iterations that are required before all tasks are complete. We define N_{α} and N_{β} to be the number of processors initially assigned α and β tasks, respectively. Assuming each processor consumes a single task per iteration, the number of tasks consumed by an overloaded processor per round (after time T_{β}) is given by $\lfloor N_{\beta}/N_{\alpha} \rfloor + 1$, or the number of tasks donated to load balancing plus the one task consumed by the processor itself. The upper bound on the number of load balancing iterations is determined by subtracting the number of migrated tasks from the number of initially allocated tasks. This, in turn, is used to compute an upper bound on the execution time of the dominating processor. A similar procedure is used to calculate T_{work} for an initially underloaded processor.

4.2. Preemptive Polling Thread Component

PREMA's preemptive polling thread adds a fixed percentage of overhead to each task. $T_{quantum}$ defines the period after which the polling thread will awaken, and is input to the model. Another input, T_{poll} is the amount of time required to complete a single polling operation, and is independent of $T_{quantum}$. With this information, the overhead attributable to the polling thread (T_{thread}) is specified as the number of thread invocations during a period of work ($T_{work}/T_{quantum}$) multiplied by the overhead per thread invocation ($2 \times T_{ctx} + T_{poll}$, where T_{ctx} is the time required for a thread context switch).

2 Due to the unpredictable nature of adaptive codes, neither the runtime system nor the application knows the location of particular tasks in advance.

4.3. Application Communication Component

The T^{app}_{comm} component describes the cost of interprocessor communication invoked by the application itself. The number and size of messages sent by each task are fixed and input to the model. Message passing (for both the application and runtime system) are modeled as a startup cost plus a cost per byte, which are also parameters of the model. Although we do not overlap communication operations, it would be trivial to do so. We also assume there is no overlapping of computation with communication since we are interested in computing an upper bound on T^{app}_{comm}.

The communication cost per task is defined as the cost per message multiplied by the number of messages a task sends. The total application communication cost is the single-task messaging cost multiplied by the number of tasks on a particular processor. Note that the number of tasks per processor will deviate from the initial assignments due to load balancing, as described in Section 4.1.

4.4. Load Balancing Communication Component

T^{lb}_{comm} defines the cost of information gathering during load balancing. With the Diffusion load balancing method, underloaded processors will send a message to each processor in the local *neighborhood*, requesting the number of tasks available for migration. If no tasks are available, new neighbors are selected and the process is repeated. Because it is not possible to predict the number of unsuccessful requests, the number of migratable tasks will serve as an optimistic lower bound.

In the case of a sink processor (one that receives tasks during load balancing), the time to request information from neighboring nodes can be expressed as the number of neighbors multiplied by the cost of sending a single request. This can be trivially altered in the case in which communication operations may be overlapped. The load balancing communication cost per task migration can then be expressed as the sum of the time to send a request (formulated using the linear message cost model previously described), the expected time on the receiver before the preemptive polling thread awakens to process the request ($T_{quantum}/2$), the time to process the request (which is input to the model), the time to send a reply (again using the linear messaging cost model), and the time to process the reply on the originating processor (an input to the model). These times constitute the "turn-around" time for a load balancing message and will be dominated by the preemptive polling thread's quantum. The quantum therefore represents a tradeoff: a shorter quantum reduces the turn-around time but adds to the overhead incurred by the runtime system.

In the case of Diffusion load balancing, no information is gathered by the source processors, so this term contributes nothing to the predicted execution time.

4.5. Load Balancing Migration Component

The T^{lb}_{migr} term of Equation 6 represents the time required for task migration, and can be broken into the cost of moving a task to a processor and the cost of receiving a task from a processor. Initially overloaded processors are charged with the cost of uninstalling, packing, and transporting a task. Conversely, initially underloaded processors must unpack and install the migrated tasks.

The costs associated with packing, unpacking, installing, and uninstalling tasks are measured quantities and are provided as inputs to the model. The cost of message passing is calculated using the linear cost model described previously.

4.6. Migration Decision Making Component

The $T^{lb}_{decision}$ term of Equation 6 represents the time required for the load balancing scheduling software to select a partner processor once it has received replies to all neighborhood information queries. This is a measured quantity which is input to the model and dependent on the scheduling policy used. For this paper, we have measured the time required by the Diffusion scheduling policy to be approximately 0.0001 sec. on a 333 MHz UltraSPARC IIi processor.

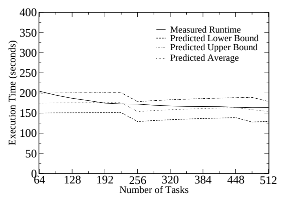
4.7. Overlap Between Components

Certain system architectures offer the capability to overlap components of the runtime model. For instance, it may be possible to off-load communication to a separate processor which manages the network interface. Another example is a multi-processor machine on which the preemptive PREMA polling thread may be allocated to a separate processor from the primary application.

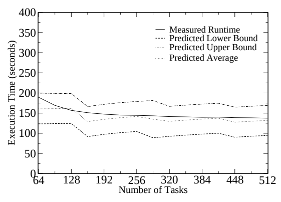
In such a case, this overlap must be removed from the overall predicted runtime. However, on the system on which our experiments were conducted, overlap was not possible.

5. Validation of the Analytic Model

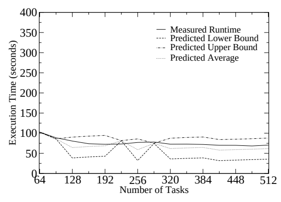
We validate the accuracy of our analytic model using both a benchmark program and an implementation of a 2D Parallel Constrained Delaunay Triangulation (PCDT) mesh refinement algorithm. We present results from three tests using the benchmark program. In the first, task execution times vary linearly from a minimum value to a maximum of twice the minimum (*linear-2* test). In the sec-



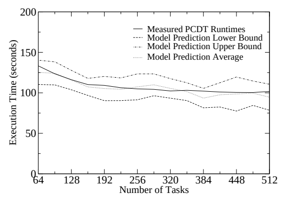
(a) 32 proc *linear-2* test



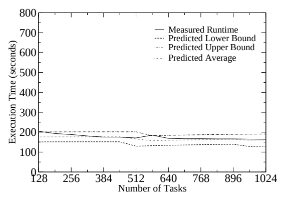
(b) 32 proc *linear-4* test



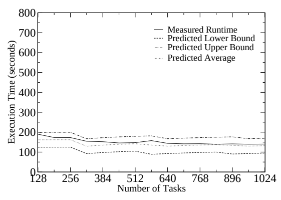
(c) 32 proc *step* test



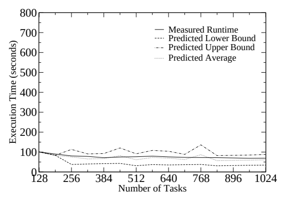
(g) 32 processor PCDT



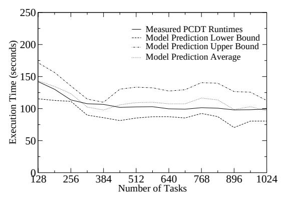
(d) 64 proc *linear-2* test



(e) 64 proc *linear-4* test



(f) 64 proc *step* test



(h) 64 processor PCDT

Figure 1. Comparison between measured benchmark run times and model predictions for 32 ((a)–(c)) and 64 ((d)–(f)) processors, and PCDT applications on 32 and 64 processors.

ond, the maximum value is a factor of four times the minimum (*linear-4* test). In the third, 25% of the tasks have the heavier weight and require double the computation time of the remaining 75% (*step* test). Such a benchmark (which does not incorporate inter-task communication) is representative of a 3D Parallel Advancing Front (PAFT) mesh generation and refinement application [8] developed within our research group. PAFT begins by partitioning a 3D domain into sub-domains and constructing triangular surface meshes for each, ensuring that the surface meshes are consistent between neighboring regions. Tetrahedralization can then progress in each sub-domain independently, with no communication required until the global mesh is reassembled before termination of the PAFT. Load imbalance arises due to varying complexity of sub-domain geometry, or the existence of "features of interest" which require mesh refinement to a higher degree of fidelity.

Figure 1 contains the results of all three tests (linear-2, linear-4, and step) on 32 and 64 homogeneous processors[3]. In each test, the granularity of the task decomposition varies with the number of tasks allocated to each processor (from 2 to 16). Each graph displays the measured program execution time, along with an upper bound, lower bound, and average prediction generated by the analytic model.

3 Our test platform consisted of 64 single-CPU (333 MHz) Sun Ultra 5 workstations with 256 MByte local memory and connected with 100 Mbit fast ethernet. The communication infrastructure on which the PREMA runtime environment was built consisted of the LAM implementation of MPI. The cluster is utilized in single-user mode through the PBS batch system; our application did not have to contend with other applications for resources.

The average prediction for the *linear-2* and *linear-4* tests differ from the measured run times by 4% or less on both 32 and 64 processors, while the error increases to roughly 10% in the case of the *step* test. This discrepancy can be explained by the smaller total execution time, and the existence of a few outlying points. For longer running programs, the error percentage decreases; this is desirable as it is these long running applications in which we are most interested.

Modeling the PCDT program is challenging for two reasons. First, the load imbalance is generated by a non-linear "heavy-tailed" task distribution. Second, tasks communicate with one another during runtime, and this behavior must be captured by the model. Figure 1(g) and Figure 1(h) indicate the effectiveness of the model on 32 and 64 processors, with an average error on 32 processors of 3.2%. On 64 processors this figure increases slightly to 6%.

6. Parametric Studies

The validated analytic model can be used to study the impact certain runtime parameters have on load balancing and application performance. The specific variables in which we are interested include the preemption quantum, number of processors, task granularity, load balancing neighborhood size, and communication latency. We begin with a look at applications exhibiting a simple bi-modal imbalance, and progress to more complex applications with a linear imbalance and inter-task communication. Finally, we will examine the effect of communication latency.

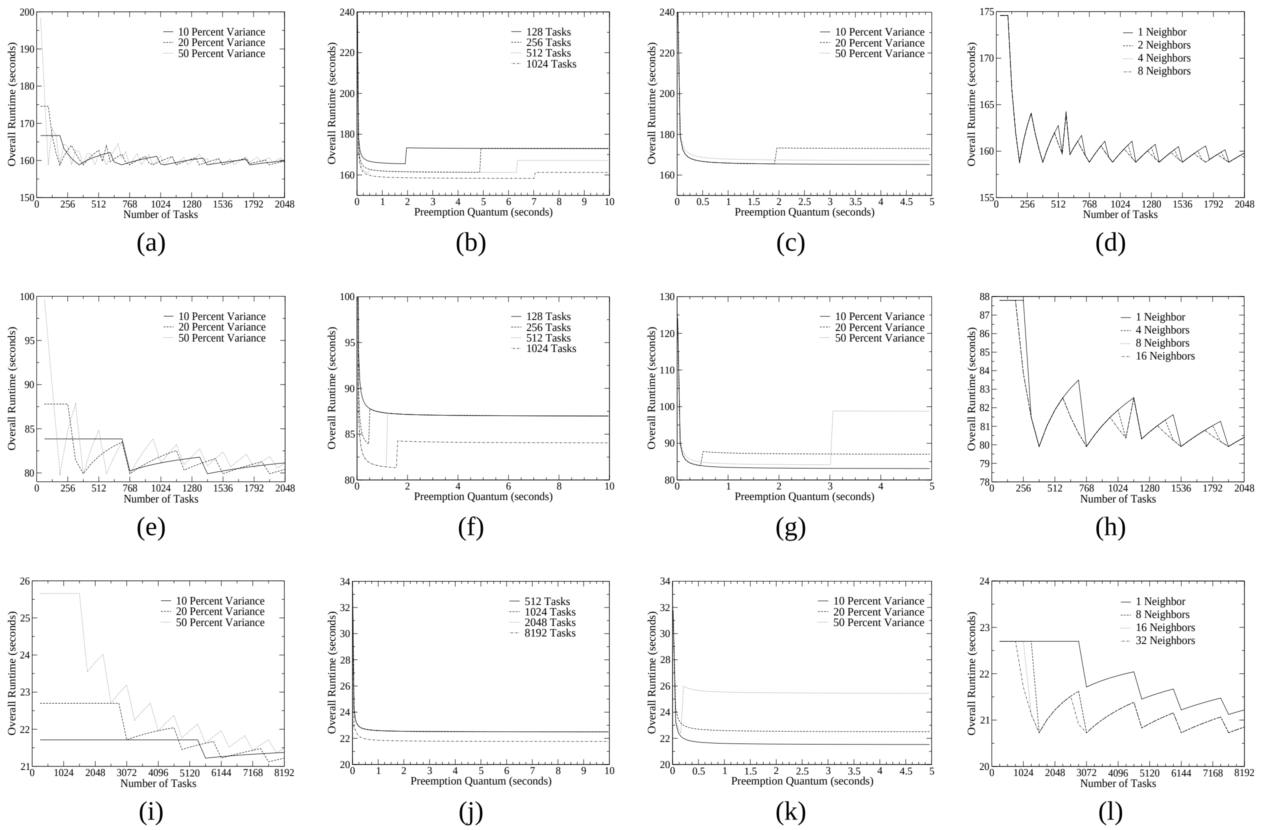


Figure 2. Bi-modal imbalance predictions on 32 (top row), 64 (middle row), and 256 (bottom row) processors.

6.1. Bi-modal Imbalance

Applications exhibiting bi-modal imbalance are composed of two task types. For this benchmark, heavy tasks make up 50% of the task count, and the *variance*, or difference in execution time between heavy and light tasks is specified at execution time. We first study how task granularity, or level of *over-decomposition*, affects overall runtime (Figure 2, column 1). Initially, increasing the number of tasks leads to a decrease in overall runtime as the load balancer has a greater deal of flexibility in task migration. A dampening periodic behavior can be seen as the number of tasks increases. The period of this effect depends on both the number of processors and the initial level of imbalance, and results from a situation in which the smoothest possible load distribution creates a workload difference between processors of almost an entire task. Further over-decomposition eliminates this effect by breaking the original task into pieces, each of which may migrate independently.

Our next experiment studies the effect the preemption quantum has on total execution time. Figure 2 (columns 2 and 3) indicates there is a range of values that will minimize run times. Quanta values which are too small lead to excessive polling thread overhead, while those that are too large cause lengthly delay in load balancing response time. In the case of large processor configurations and large task variance (Figure 2(k)), the range of optimal quanta values is quite small.

Finally, in Figure 2 column 4, we examine the impact of load balancing neighborhood size (the neighborhood defines the set of processors with whom load balancing information is exchanged). As the number of processors grows, the time required to probe all processors for tasks also increases. As a result, some tasks that are candidates for migration may not be located, and thus load balancing efficiency degrades. Increasing the number of neighbors can overcome this problem (Figure 2(l)). However, for small processor configurations, this is of little benefit.

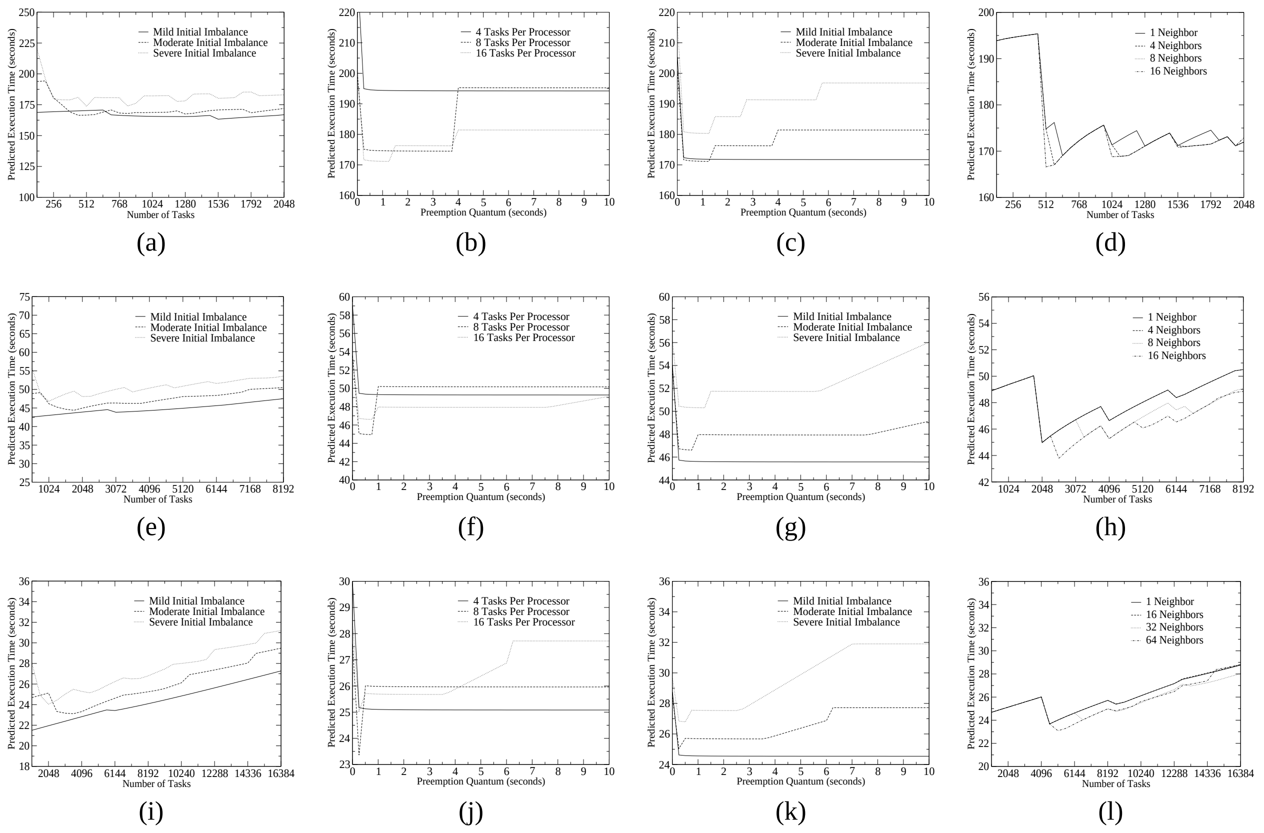


Figure 3. Linear imbalance data on 64 (top row), 256 (middle row), and 512 (bottom row) processors.

6.2. Linear Imbalance

In our second set of experiments, we study a more complex application type in which task weights are distributed linearly over one of three ranges. *Mild* imbalance varies task execution times over a range in which the heaviest tasks require 20% more time than the lightest ones. With *moderate* imbalance, heavy tasks are twice as costly as the lightest ones, while *severe* imbalance increases the range to a factor of four. Secondly, each task has four “neighbors” with whom it communicates during its execution. This is a common communication pattern when, for instance, processors are arranged in a logical 2D grid.

The effect of over-decomposition is shown in Figure 3, column 1. In this case, the flexibility given to the load balancer by a fine task granularity is in tension with the greater amount of required inter-task communication. This tension will eventually penalize greater levels of over-decomposition, particularly in the case of a mild initial imbalance. In the cases of moderate and severe initial imbalance, load balancing will produce a minimum execution time, after which point further over-decomposition increases application run time.

The impact of the preemption quantum (Figure 3, column 2) indicates, as before, there is an optimal range of quantum values which lead to a minimum execution time. The size of this optimal range tends to decrease as the number of processors grows. Repeating the same experiment with various levels of initial imbalance (Figure 3, column 3) indicates that this range remains roughly constant, regardless of the degree of imbalance. However, a finer task granularity allows the load balancer to be more tolerant of larger quantum values.

Lastly, we examine the impact of neighborhood size, and see that the results are consistent with observations derived from experiments with a bi-modal initial imbalance (Figure 3, column 4).

7. PREMA Load Balancing Performance

The power of the analytic model's predictive capability lies in its ability to generate optimal values for the configuration of the PREMA runtime software. Comparisons between PREMA and prevalent load balancing tools found in the research community demonstrate performance improvements on a set of synthetic micro-benchmarks. While we were able to configure PREMA off-line using the analytic modeling technique we have described, obtaining optimal results with other tools required trial-and-error experimentation and repeated benchmark runs.

As a basis for the evaluation of PREMA's runtime efficiency, we present the results from three software tools widely used by the research community: the Metis [18] library of repartitioning tools, the iterative load balancers incorporated in to the Charm++ library [16, 17], and Charm++'s seed-based load balancers [5]. The first two are designed for load balancing loosely synchronous applications; by comparing their performance with that of the PREMA system, we demonstrate that the loosely synchronous model is not appropriate for efficiently supporting asynchronous applications. The seed-based balancers that are part of the Charm++ runtime library are themselves asynchronous, and therefore serve as a basis for evaluating the efficiency of our software implementation.

The benchmark program creates a set of discrete tasks that do not communicate during their execution, similar to the PAFT application described in Section 5. A heavy weight is assigned to 10% of the tasks, and a lighter weight (half of the heavy) to the remaining 90%. We have used predictions generated from our model to set the number of tasks per processor to 8, and the preemption quantum to 0.5 seconds. While a more even load diffusion is possible with a finer task granularity, the predictions generated from the model indicate that this will not lead to significant improvement in overall run time.

Figure 4 contains the results of our experiments run on 64 processors. Compared with no load balancing, PREMA provides an overall performance improvement of 38% (Figures 4 (a) and (b)). When using Metis, processors must synchronize in order to calculate a new partitioning. The benchmark program refrains from synchronization until a particular processor's local load level drops below a predefined threshold[4], at which point a synchronization request is broadcast to all processors. This message may arrive during the processing of a task, in which case it will not be processed until the task is complete. Due to this synchronization overhead imposed by Metis, PREMA is able to provide a performance improvement of 40%. However, when the percentage of "heavy" tasks increases from 10% to 25%, Metis is able to more evenly distribute the load. Synchronization overhead associated with Metis still imposes a significant penalty, and PREMA is able to provide a 39% performance improvement.

PREMA provides a similar improvement over the loosely synchronous load balancing provided by Charm++'s iterative balancers (Figure 4(f)), which synchronize processors after a certain number of tasks have been executed. Using measurements taken during the previous iteration, tasks may be migrated under the assumption that computation in the next iteration will proceed in a similar fashion. Experimentally, we have found that four load balancing iterations provide the best trade-off between load balancing quality and synchronization overhead. However, even in this case PREMA provides a runtime improvement of 41%.

A more interesting case is Charm++'s asynchronous, seed-based balancing capability (Figure 4(g)). We can see that this method is more successful than either loosely synchronous method at distributing the work load. However, the number of idle cycles on each processor are evidence of overhead incurred by the runtime system. As a result, PREMA is able to provide a performance improvement of 20%.

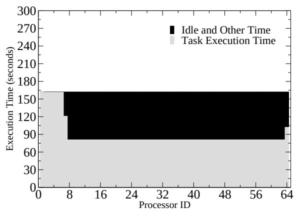
Lastly, we have conducted an experiment in which the analytic model is used in conjunction with the PREMA software to tune the performance of a Parallel Constrained Delaunay Triangulation (PCDT) [9, 10] program. On 64 processors, the model predicted a performance improvement of 3.6% would be possible with a task granularity of 16 versus 8 tasks per processor. In our experiments, we observed a performance improvement of 3.4%, with our predictions differing from the measured execution time by 2%. In Figure 4, we present our results using the finer task granularity.

In Figure 4 (c) and (d), we show that PREMA is able to provide an improvement in execution time of 19% over no load balancing. Although we do not have the resources to implement the PCDT application using all four load balancing toolkits, we see that our results with PREMA are consistent with our observations in the previous set of experiments using a synthetic microbenchmark.

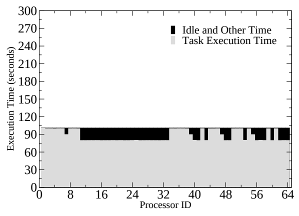
8. Related Work

The modeling of dynamic load balancing schemes is typically based on one of four methods. The first requires applications to conform to a well-understood parallel computing model, such as Bulk Synchronous Processing (BSP) [7]. The work by Nyland, et al. [23] is one such example, in which the BSP model is used to estimate the costs of spatial decomposition algorithms in the context of Molecular Dy-

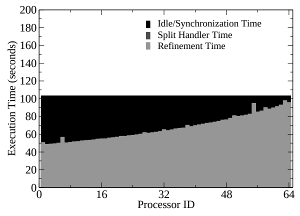
4 The threshold for all load balancing methods is identical.



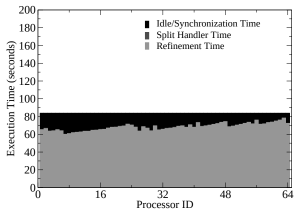
(a) Benchmark – No LB



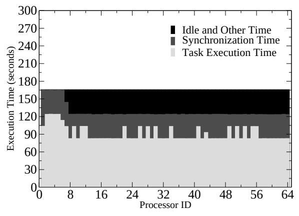
(b) Benchmark – PREMA



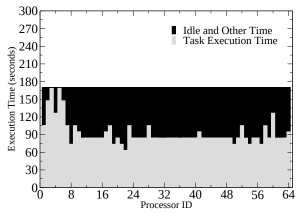
(c) PCDT – No LB



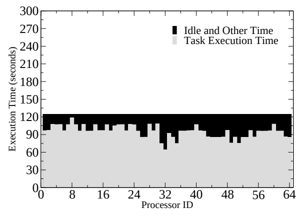
(d) PCDT – PREMA



(e) Benchmark
Metis



(f) Benchmark
Charm++ (Iter.)



(g) Benchmark
Charm++ (Async.)

Figure 4. Benchmark and PCDT performance on 64 processors

namics codes. However, the irregular applications in which we are interested are not made up of distinct phases of computation, and therefore do not conform to these models, making other techniques necessary.

The second category involves simulation. In [25], the authors use a workload generator to compare the performance of different load balancing techniques. The authors of [1] use a large simulation data set to train a neural network which predicts load balancing performance under various system parameters (time to transfer a task, time to gather load balancing information, and time to exchange load information between nodes).

A third group of research uses queueing theory or Petri nets to perform analyses. Some examples can be found in [6, 22, 13]. Such work may address topics in which we are also interested, such as the impact of communication latency on load balancing, comparison between static and dynamic load balancing policies, and the assignment of tasks to processing elements. However, the computational requirements for solving the potentially large systems of equations associated with the underlying Markovian processes make this approach less practical for the type of large-scale parametric studies we wish to undertake. In addition, as one of the future goals of our research is to implement adaptive application steering through real-time, online modeling feedback, such a time consuming approach is infeasible.

Other researchers use analytic techniques to predict load balancing performance. Ghosh, et al. [15] derive techniques useful for predicting the degree of imbalance remaining after a particular load balancing cycle. Work done at Los Alamos National Laboratory [20, 19, 21] and at the San Diego Supercomputing Center [24] have developed analytic models similar to what we describe here for the purposes of predicting and evaluating application performance on newly installed parallel machines or for predicting end-to-end application performance [12]. While we follow similar techniques in developing our model, we tackle the challenging problem of modeling dynamic and adaptive load balancing.

9. Conclusions

We have outlined an analytic technique which allows us to model applications with generalized task weight distributions in the presence of dynamic load balancing. We are then able to use the model in conjunction with the PREMA runtime toolkit to optimize application performance. We have demonstrated the effectiveness of our modeling and software technology using a synthetic micro-benchmark and a parallel mesh generation application. Through a comparison with several tools prevalent in the field today, we show that our methodology provides significant performance benefits to the user.

10. Acknowledgements

This work was performed [in part] using the Sciclone computational facilities at the College of William and Mary. In addition, we wish to thank Andrey Chernikov for his work in developing the Parallel Constrained Delaunay Triangulation program used in this work.

References

[1] I. Ahmad, A. Ghafoor, K. Mehrotra, and C. Mohan. Performance modeling of load balancing algorithms using neural networks. *Concurrency; Practice and Experience*, 6(5):393–409, 1994.

[2] K. Barker. *Runtime Support for Load Balancing of Parallel Adaptive and Irregular Applications*. PhD thesis, College of William and Mary, April 2004.

[3] K. Barker and N. Chrisochoides. An evaluation of a framework for the dynamic load balancing of highly adaptive and irregular applications. In *Proc. of the IEEE/ACM SC'03*, 2003.

[4] K. Barker, N. Chrisochoides, and K. Pingali. A load balancing framework for adaptive and asynchronous applications. *IEEE Trans. on Parallel and Distributed Computing*, 15(2):77–101, February 2004.

[5] J. Booth. Balancing priorities and load for state space search on large parallel machines. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[6] A. Brunstrom and R. Simha. Dynamic versus static load balancing in a pipeline computation. *Intern. Journal of Modeling and Simulation*, 17(4):317–327, 1997.

[7] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk synchronous parallel computing – a paradigm for transportable software. In *Proc. of the 28th Annual Hawaii Conference on System Sciences*, volume II. IEEE Computer Society Press, January 1995.

[8] A. Chernikov, N. Chrisochoides, and K. Barker. Parallel programming environment for mesh generation. In *Proceedings of 8th International Conference on Numerical Grid Generation*, Honolulu, Hawaii, 2002.

[9] L. Chew. Constrained delaunay triangulations. *Algorihmica*, 4:97–108, 1989.

[10] N. Chrisochoides, P. Chew, and F. Sukup. Parallel constrained delaunay meshing. In *1997 Symposium on Trends in Unstructured Mesh Generation*, pages 89–96, June 1997.

[11] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989.

[12] E. Deelman, A. Dube, A. Hoisie, Y. Luo, R. Oliver, D. Sundaram-Stukel, and H. Wasserman. Poems: End-to-end performance design of large parallel adaptive compuational systems. In *Proc. of the First International Workshop on Software Performance*, pages 18–30, October 1998.

[13] R. Esser, J. Janneck, and M. Naedele. Applying an object-oriented petri net language to heterogeneous systems design. In *Proc. of the Workshop on Petri Nets in Systems Engineering*, September 1997.

[14] W. Gautschi. *Numerical Analysis: An Introduction*. Birkhäuser, Boston, 1997.

[15] B. Ghosh, F. Leighton, B. Maggs, S. Muthukrishnan, C. Plaxton, R. Rajaraman, A. Richa, R. Tarjan, and D. Zuckerman. Tight analyses of two local load balancing algorithms. In *Proc. of the 27th Annual ACM Symp. on Theory of Comput.*, pages 548–558, May 1995.

[16] L. Kalé, M. Bhandarkar, and R. Brunner. Load balancing in parallel molecular dynamics. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, volume 1457 of *Lecture Notes in Computer Science*, pages 251–??, 1998.

[17] L. Kalé, M. Bhandarkar, and R. Brunner. Run-time support for adaptive load balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun, Mexico*, volume 1800, pages 1152–1159, March 2000.

[18] G. Karypis and V. Kumar. ParMETIS: Parallel graph paritioning and sparse matrix ordering library. Technical Report 97-060, Dept. of Computer Science, Univ. of Minnesota, 1997.

[19] D. Kerbyson, A. Hoisie, and H. Wasserman. Modeling the performance of large-scale systems. *IEEE Proceedings on Software*, 150(4):214–221, August 2003.

[20] D. Kerbyson, A. Hoisie, and H. Wasserman. *Verifying Large-Scale System Performance During Installation Using Modeling*. Kluwer, September 2003.

[21] D. Kerbyson, A. Hoisie, and H. Wasserman. Use of predictive performance modeling during large-scale system installation. *To appear in Parallel Processing Letters*, 2005.

[22] M. Mitzenmacher. On the analysis of randomized load balancing schemes. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 292–301, 1997.

[23] L. Nyland, J. Prins, R. H. Yun, J. Hermans, H. Kum, and L. Wang. Modeling dynamic load balancing in molecular dynamics to achieve scalable parallel execution. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 356–365, 1998.

[24] A. Snavely, N. Wolter, and L. Cartington. Modeling application performance by convolving machine signatures with application profi les. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, Texas, December 2001.

[25] C. Xu, B. Monien, R. Lüling, and F. Lau. An analytical comparison of nearest neighbor algorithms for load balancing in parallel computers. In *Proc. of 9th Intern. Parallel Processing Symposium*, pages 472–479, 1995.